Q1. Explain the concept of a prefix sum array and its applications. Provide examples to support your answer.

A **prefix sum array** is a derived array in which each element at index i represents the sum of all elements from index 0 to i in the original array. In other words, for a given array arr[], the prefix sum array prefix[] is defined as:

prefix[i]=arr[0]+arr[1]+...+arr[i]prefix[i] = arr[0] + arr[1] + ... + arr[i]prefix[i]=arr[0]+arr[1]+...+arr[i]

How to Construct a Prefix Sum Array

For an array arr[] of size n, the prefix sum array can be built in O(n) time as:

java

```
int[] prefix = new int[n];

prefix[0] = arr[0];

for (int i = 1; i < n; i++) {

   prefix[i] = prefix[i - 1] + arr[i];

}
```

Applications of Prefix Sum Array

1. Efficient Range Sum Queries

Instead of summing elements from l to r repeatedly (which takes O(n) time), you can do it in O(1) using:

sum(l,r)=prefix[r]−prefix[l−1]\text{sum}(l, r) = prefix[r] - prefix[l - 1]sum(l,r)=prefix[r]−prefix[l−1]

(If l = 0, then sum = prefix[r])

2. Solving Subarray Problems

Prefix sums help in finding:

- Maximum sum subarrays

- Number of subarrays with a certain sum
- Zero-sum subarrays

3. Frequency Count and Range Updates

In problems involving multiple range updates or queries (like adding a value x to all elements in a range), a variant called **difference array** (inverse of prefix sum) is used.

4. 2D Prefix Sum (Matrix)

Used to efficiently calculate the sum of elements in a sub-matrix.

5. Real-World Use Cases

- Game development (tracking scores or health over time)
- Analytics platforms (aggregating data over a time window)
- Financial systems (calculating cumulative profit/loss)

Q2. Program to find sum of elements in range [L, R] using prefix sum array

```
class PrefixSum {
   int[] prefix;

   PrefixSum(int[] arr) {
      prefix = new int[arr.length];
      prefix[0] = arr[0];
      for (int i = 1; i < arr.length; i++) {
         prefix[i] = prefix[i - 1] + arr[i];
      }
   }
```

```
    int rangeSum(int L, int R) {
        if (L == 0) return prefix[R];
        return prefix[R] - prefix[L - 1];
    }
}
```

Algorithm:

1. Create a prefix array.
2. Fill prefix[i] = prefix[i-1] + arr[i].
3. For range [L, R], return prefix[R] - prefix[L-1].

Time: O(n) for preprocessing, O(1) per query

Space: O(n)

Q3. Find equilibrium index in array

```
class EquilibriumIndex {
    static int findEquilibrium(int[] arr) {
        int total = 0, leftSum = 0;
        for (int num : arr) total += num;
        for (int i = 0; i < arr.length; i++) {
            total -= arr[i];
            if (leftSum == total) return i;
            leftSum += arr[i];
        }
        return -1;
    }
```

```
}
```

Algorithm:

1. Find total sum.

2. Traverse and maintain left sum.

3. Check if left sum equals total sum - current element.

Time: O(n), Space: O(1)

Q4. Check if array can be split into two parts with equal prefix and suffix sums

```
class SplitEqualSum {
    static boolean canSplit(int[] arr) {
        int total = 0, prefixSum = 0;
        for (int num : arr) total += num;
        for (int i = 0; i < arr.length - 1; i++) {
            prefixSum += arr[i];
            if (prefixSum == total - prefixSum) return true;
        }
        return false;
    }
}
```

Time: O(n), Space: O(1)

Q5. Maximum sum of subarray of size K

```
class MaxSubarraySumK {
   static int maxSum(int[] arr, int k) {
      int windowSum = 0, maxSum = 0;
      for (int i = 0; i < k; i++) windowSum += arr[i];
      maxSum = windowSum;
      for (int i = k; i < arr.length; i++) {
         windowSum += arr[i] - arr[i - k];
         maxSum = Math.max(maxSum, windowSum);
      }
      return maxSum;
   }
}
```

Sliding window used

Time: O(n), Space: O(1)

Q6. Length of the longest substring without repeating characters

```
class LongestUniqueSubstring {
   static int lengthOfLongestSubstring(String s) {
      int[] lastIndex = new int[256];
      for (int i = 0; i < 256; i++) lastIndex[i] = -1;
      int maxLength = 0, start = 0;

      for (int end = 0; end < s.length(); end++) {
```

```
        if (lastIndex[s.charAt(end)] >= start) {
            start = lastIndex[s.charAt(end)] + 1;
        }
        lastIndex[s.charAt(end)] = end;
        maxLength = Math.max(maxLength, end - start + 1);
    }
    return maxLength;
  }
}
```

Algorithm:

1. Use a sliding window with a map to store last seen index.

2. Update window start when a repeating character is found.

Time: O(n), Space: O(1) [for fixed ASCII charset]

Example: "abcabcbb" -> 3

Q7. Explain the sliding window technique and its use in string problems

Sliding window is a technique where we maintain a subset (window) of the data and slide it across the array/string.

Useful for problems requiring contiguous subarrays or substrings.

Examples:

- Longest substring without repeating characters

- Max sum of subarray of size K

- Anagram detection in string

Time Complexity: O(n), Space: O(1)/O(k)

Q8. Longest palindromic substring

```
class LongestPalindrome {
   static String longestPalindrome(String s) {
      if (s.length() < 1) return "";
      int start = 0, end = 0;
      for (int i = 0; i < s.length(); i++) {
         int len1 = expandFromMiddle(s, i, i);
         int len2 = expandFromMiddle(s, i, i + 1);
         int len = Math.max(len1, len2);
         if (len > end - start) {
            start = i - (len - 1) / 2;
            end = i + len / 2;
         }
      }
      return s.substring(start, end + 1);
   }

   static int expandFromMiddle(String s, int left, int right) {
      while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
         left--;
         right++;
      }
      return right - left - 1;
   }
}
```

Algorithm: Expand Around Center

Time: O(n^2), Space: O(1)

Example: "babad" -> "bab" or "aba"

Q9. Longest common prefix among strings

```
class LongestCommonPrefix {
   static String longestCommonPrefix(String[] strs) {
      if (strs.length == 0) return "";
      String prefix = strs[0];
      for (int i = 1; i < strs.length; i++) {
         while (strs[i].indexOf(prefix) != 0) {
            prefix = prefix.substring(0, prefix.length() - 1);
            if (prefix.isEmpty()) return "";
         }
      }
      return prefix;
   }
}
```

Algorithm:

1. Start with first string as prefix.

2. Trim prefix if next string doesn't start with it.

Time: O(n*m), Space: O(1)

Q10. Generate all permutations of a string

```
class StringPermutations {
   static void permute(String str, int l, int r) {
      if (l == r)
         System.out.println(str);
      else {
         for (int i = l; i <= r; i++) {
            str = swap(str, l, i);
            permute(str, l + 1, r);
            str = swap(str, l, i);
         }
      }
   }

   static String swap(String a, int i, int j) {
      char[] charArray = a.toCharArray();
      char temp = charArray[i];
      charArray[i] = charArray[j];
      charArray[j] = temp;
      return String.valueOf(charArray);
   }
}
```

Backtracking approach

Time: O(n*n!), Space: O(n)

Example: abc -> abc, acb, bac, bca, cab, cba

Q11. Find two numbers in a sorted array that add up to a target

```
class TwoSumSorted {
   static int[] twoSum(int[] nums, int target) {
      int left = 0, right = nums.length - 1;
      while (left < right) {
         int sum = nums[left] + nums[right];
         if (sum == target) return new int[]{left, right};
         if (sum < target) left++;
         else right--;
      }
      return new int[]{-1, -1};
   }
}
```

Two-pointer approach

Time: O(n), Space: O(1)

Q12. Lexicographically next greater permutation

```
class NextPermutation {
   static void nextPermutation(int[] nums) {
      int i = nums.length - 2;
      while (i >= 0 && nums[i] >= nums[i + 1]) i--;
      if (i >= 0) {
         int j = nums.length - 1;
```

```
            while (nums[j] <= nums[i]) j--;
            swap(nums, i, j);
        }
        reverse(nums, i + 1);
    }

    static void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }

    static void reverse(int[] nums, int start) {
        int end = nums.length - 1;
        while (start < end) {
            swap(nums, start++, end--);
        }
    }
}
```

Time: O(n), Space: O(1)

Q13. Merge two sorted linked lists

```
class MergeSortedLists {
    static class ListNode {
        int val;
        ListNode next;
        ListNode(int x) { val = x; }
    }

    static ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(-1), curr = dummy;
        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                curr.next = l1; l1 = l1.next;
            } else {
                curr.next = l2; l2 = l2.next;
            }
            curr = curr.next;
        }
        curr.next = (l1 != null) ? l1 : l2;
        return dummy.next;
    }
}
```

Time: O(n + m), Space: O(1)

Q14. Median of two sorted arrays using binary search

```
class MedianSortedArrays {
   static double findMedianSortedArrays(int[] nums1, int[] nums2) {
      if (nums1.length > nums2.length) return findMedianSortedArrays(nums2, nums1);

      int x = nums1.length, y = nums2.length;
      int low = 0, high = x;

      while (low <= high) {
         int partitionX = (low + high) / 2;
         int partitionY = (x + y + 1) / 2 - partitionX;

         int maxX = (partitionX == 0) ? Integer.MIN_VALUE : nums1[partitionX - 1];
         int minX = (partitionX == x) ? Integer.MAX_VALUE : nums1[partitionX];

         int maxY = (partitionY == 0) ? Integer.MIN_VALUE : nums2[partitionY - 1];
         int minY = (partitionY == y) ? Integer.MAX_VALUE : nums2[partitionY];

         if (maxX <= minY && maxY <= minX) {
            if ((x + y) % 2 == 0)
               return ((double)Math.max(maxX, maxY) + Math.min(minX, minY)) / 2;
            else
               return (double)Math.max(maxX, maxY);
         } else if (maxX > minY) {
            high = partitionX - 1;
         } else {
```

```
            low = partitionX + 1;
        }
    }
    throw new IllegalArgumentException();
  }
}
```

Time: O(log(min(n, m))), Space: O(1)

Q15. Find the k-th smallest element in a sorted matrix

```
import java.util.PriorityQueue;
class KthSmallestInMatrix {
  static class Node implements Comparable<Node> {
    int val, r, c;
    Node(int v, int r, int c) { this.val = v; this.r = r; this.c = c; }
    public int compareTo(Node o) { return this.val - o.val; }
  }

  static int kthSmallest(int[][] matrix, int k) {
    int n = matrix.length;
    PriorityQueue<Node> pq = new PriorityQueue<>();
    for (int i = 0; i < n; i++) pq.offer(new Node(matrix[i][0], i, 0));

    while (--k > 0) {
      Node node = pq.poll();
      if (node.c < n - 1)
        pq.offer(new Node(matrix[node.r][node.c + 1], node.r, node.c + 1));
    }
```

```
        return pq.peek().val;
    }
}
```

Time: O(k log n), Space: O(n)

Q16. Find the majority element (appears more than n/2 times)

```
class MajorityElement {
    static int majorityElement(int[] nums) {
        int count = 0, candidate = -1;
        for (int num : nums) {
            if (count == 0) {
                candidate = num;
                count = 1;
            } else if (candidate == num) {
                count++;
            } else {
                count--;
            }
        }
        return candidate;
    }
}
```

Moore’s Voting Algorithm

Time: O(n), Space: O(1)

Q17. Trapping Rain Water

```
class TrappingRainWater {
    static int trap(int[] height) {
        int left = 0, right = height.length - 1;
        int leftMax = 0, rightMax = 0, result = 0;
        while (left < right) {
            if (height[left] < height[right]) {
                if (height[left] >= leftMax)
                    leftMax = height[left];
                else
                    result += leftMax - height[left];
                left++;
            } else {
                if (height[right] >= rightMax)
                    rightMax = height[right];
                else
                    result += rightMax - height[right];
                right--;
            }
        }
        return result;
    }
}
```

Two-pointer approach

Time: O(n), Space: O(1)

Q18. Maximum XOR of two numbers in an array

```
class MaximumXOR {
   static int findMaximumXOR(int[] nums) {
      int max = 0, mask = 0;
      Set<Integer> set = new HashSet<>();
      for (int i = 31; i >= 0; i--) {
         mask |= (1 << i);
         set.clear();
         for (int num : nums)
            set.add(num & mask);
         int temp = max | (1 << i);
         for (int prefix : set) {
            if (set.contains(prefix ^ temp)) {
               max = temp;
               break;
            }
         }
      }
      return max;
   }
}
```

Time: O(n), Space: O(n)

Q19. Maximum Product Subarray

```
class MaxProductSubarray {
    static int maxProduct(int[] nums) {
        int maxProd = nums[0], minProd = nums[0], result = nums[0];
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] < 0) {
                int temp = maxProd;
                maxProd = minProd;
                minProd = temp;
            }
            maxProd = Math.max(nums[i], maxProd * nums[i]);
            minProd = Math.min(nums[i], minProd * nums[i]);
            result = Math.max(result, maxProd);
        }
        return result;
    }
}
```

Time: O(n), Space: O(1)

Q20. Count numbers with unique digits (for given number of digits)

```
class CountUniqueDigits {
    static int countNumbersWithUniqueDigits(int n) {
        if (n == 0) return 1;
        int res = 10, uniqueDigits = 9, available = 9;
```

```
        while (n-- > 1 && available > 0) {
            uniqueDigits *= available;
            res += uniqueDigits;
            available--;
        }
        return res;
    }
}
```

Time: O(n), Space: O(1)

Q21. Count number of 1s in binary representation from 0 to n

```
class CountBits {
    static int[] countBits(int n) {
        int[] res = new int[n + 1];
        for (int i = 1; i <= n; i++) {
            res[i] = res[i >> 1] + (i & 1);
        }
        return res;
    }
}
```

Time: O(n), Space: O(n)

Q22. Check if a number is power of two using bit manipulation

```
class PowerOfTwo {
   static boolean isPowerOfTwo(int n) {
      return n > 0 && (n & (n - 1)) == 0;
   }
}
```

Time: O(1), Space: O(1)

Q23. Maximum XOR of two numbers

```
class MaximumXOR {
   static int findMaximumXOR(int[] nums) {
      int max = 0, mask = 0;
      Set<Integer> set = new HashSet<>();
      for (int i = 31; i >= 0; i--) {
         mask |= (1 << i);
         set.clear();
         for (int num : nums)
            set.add(num & mask);
         int temp = max | (1 << i);
         for (int prefix : set) {
            if (set.contains(prefix ^ temp)) {
               max = temp;
               break;
```

```
            }
          }
        }
        return max;
    }
}
```

Time: O(n), Space: O(n)

Q24. Concept of Bit Manipulation

Bit Manipulation involves using bitwise operators (&, |, ^, ~, <<, >>) for optimizing algorithms.

Advantages:

- Faster computation
- Memory efficiency
- Useful in masks, toggling bits, and solving complex problems like subset generation.

Q25. Next greater element for each element in an array

```
class NextGreaterElement {
    static int[] nextGreaterElements(int[] nums) {
        Stack<Integer> stack = new Stack<>();
        int[] res = new int[nums.length];
        Arrays.fill(res, -1);
        for (int i = 0; i < nums.length; i++) {
```

```
            while (!stack.isEmpty() && nums[stack.peek()] < nums[i]) {
                res[stack.pop()] = nums[i];
            }
            stack.push(i);
        }
        return res;
    }
}
```

Time: O(n), Space: O(n)

Q26. Remove the n-th node from end of a singly linked list

```
class RemoveNthNode {
    static class ListNode {
        int val;
        ListNode next;
        ListNode(int x) { val = x; }
    }

    static ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode first = dummy, second = dummy;
        for (int i = 0; i <= n; i++) first = first.next;
        while (first != null) {
            first = first.next;
```

```
            second = second.next;
        }
        second.next = second.next.next;
        return dummy.next;
    }
}
```

Time: O(n), Space: O(1)

Q27. Find the node where two singly linked lists intersect

```
class LinkedListIntersection {
    static RemoveNthNode.ListNode getIntersectionNode(RemoveNthNode.ListNode headA,
RemoveNthNode.ListNode headB) {
        RemoveNthNode.ListNode a = headA, b = headB;
        while (a != b) {
            a = (a == null) ? headB : a.next;
            b = (b == null) ? headA : b.next;
        }
        return a;
    }
}
```

Time: O(n), Space: O(1)

Q28. Implement two stacks in a single array

```
class TwoStacks {
   int size, top1, top2;
   int[] arr;

   TwoStacks(int n) {
      size = n;
      arr = new int[n];
      top1 = -1;
      top2 = n;
   }

   void push1(int x) {
      if (top1 < top2 - 1) arr[++top1] = x;
   }

   void push2(int x) {
      if (top1 < top2 - 1) arr[--top2] = x;
   }

   int pop1() {
      if (top1 >= 0) return arr[top1--];
      return -1;
   }

   int pop2() {
```

```
        if (top2 < size) return arr[top2++];
        return -1;
    }
}
```

Time: O(1), Space: O(n)

Q29. Check if integer is palindrome without converting to string

```
class PalindromeNumber {
    static boolean isPalindrome(int x) {
        if (x < 0 || (x % 10 == 0 && x != 0)) return false;
        int reversed = 0;
        while (x > reversed) {
            reversed = reversed * 10 + x % 10;
            x /= 10;
        }
        return x == reversed || x == reversed / 10;
    }
}
```

Time: O(log n), Space: O(1)

Q30. Explain Linked Lists and Their Applications. Provide examples to support your answer.

A Linked List is a linear data structure in which elements, called nodes, are stored in separate memory locations. Each node contains two parts:

1. Data – the value stored in the node.
2. Pointer (or link) – a reference to the next node in the sequence.

Unlike arrays, linked lists do not require contiguous memory allocation. This makes them dynamic and flexible in memory usage.

Types of Linked Lists:

- Singly Linked List – each node points to the next node.
- Doubly Linked List – each node points to both next and previous nodes.
- Circular Linked List – last node points back to the first node.

Key Features

- Dynamic memory allocation – no need to declare size beforehand.
- Efficient insertion/deletion – especially at the beginning or middle.
- No memory wastage – grows or shrinks as needed.

Applications of Linked Lists

1. Dynamic Memory Allocation

Linked lists are ideal for allocating memory on-the-fly. For example, in operating systems, the kernel uses linked lists to manage memory blocks.

2. Efficient Insertions and Deletions

In arrays, inserting or deleting elements requires shifting, which takes O(n) time. Linked lists perform these operations in O(1) time if the pointer to the node is available.

3. Implementation of Data Structures

- Stacks – using a singly linked list where insertion and deletion happen at the head.
- Queues – with insertion at the tail and deletion from the head.

- Hash Tables – for handling collisions via chaining (linked lists).
- Graphs – adjacency list representation uses linked lists for storing neighboring vertices.

4. Polynomial Arithmetic

Linked lists can represent polynomials where each node holds the coefficient and exponent. This is useful in symbolic algebra and compiler design.

5. Music or Image Viewer Applications

Doubly linked lists allow forward and backward navigation between songs/images.

6. Real-World Use Cases

- Undo functionality in text editors (implemented using stacks with linked lists).
- Browser history (back and forward functionality using doubly linked lists).
- Job scheduling systems (using circular linked lists for round-robin scheduling).

Q31. Maximum in every sliding window of size K using deque

```
class MaxSlidingWindow {
   static int[] maxSlidingWindow(int[] nums, int k) {
      Deque<Integer> deque = new LinkedList<>();
      int[] result = new int[nums.length - k + 1];
      for (int i = 0; i < nums.length; i++) {
         while (!deque.isEmpty() && deque.peek() < i - k + 1) deque.poll();
         while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) deque.pollLast();
         deque.offer(i);
         if (i >= k - 1) result[i - k + 1] = nums[deque.peek()];
      }
      return result;
   }
}
```

Time: O(n), Space: O(k)

Q32. Largest rectangle in histogram

```
class LargestRectangleHistogram {
    static int largestRectangleArea(int[] heights) {
        Stack<Integer> stack = new Stack<>();
        int maxArea = 0;
        for (int i = 0; i <= heights.length; i++) {
            int h = (i == heights.length) ? 0 : heights[i];
            while (!stack.isEmpty() && h < heights[stack.peek()]) {
                int height = heights[stack.pop()];
                int width = stack.isEmpty() ? i : i - stack.peek() - 1;
                maxArea = Math.max(maxArea, height * width);
            }
            stack.push(i);
        }
        return maxArea;
    }
}
```

Time: O(n), Space: O(n)

Q33. Explain the Sliding Window Technique in Array Problems. Provide examples and applications.

The Sliding Window Technique is a powerful method used to optimize problems that involve checking or computing values for all contiguous subarrays or substrings of a fixed or variable size in an array or string.

Instead of recalculating the result for every subarray (which usually takes O(n × k) time), sliding window reduces the time complexity to O(n) by maintaining a running window of size k and updating it as the window moves through the array.

How It Works:

1. Define the start and end of the window (usually with two pointers).
2. Process the elements inside the window.
3. Slide the window forward by incrementing the pointers.
4. Update results accordingly without re-processing the whole subarray.

Types of Sliding Window:

- Fixed Size Window: Used when the subarray size is fixed.
- Variable Size Window: Used when the window size changes based on a condition (e.g., sum, character frequency, etc.).

Time and Space Complexity

- Time: O(n) – every element is processed at most twice (entering and exiting the window).
- Space: O(k) or less, depending on the problem (e.g., using hash maps, deques, sets).

Applications of Sliding Window

1. Maximum / Minimum Sum Subarray of Size K

Find the maximum sum of any contiguous subarray of size k.

```
int maxSum = 0, windowSum = 0;
for (int i = 0; i < k; i++) windowSum += arr[i];
for (int i = k; i < n; i++) {
   windowSum += arr[i] - arr[i - k];
   maxSum = Math.max(maxSum, windowSum);
}
```

2. Counting Distinct Elements in a Window

Use a hash map or set to count distinct elements in each window of size k.

3. Longest Substring Without Repeating Characters

A common sliding window problem in strings using a dynamic window and hash set.

4. Smallest Subarray with Sum ≥ X

Variable window size problem used in optimization questions.

5. Binary Subarrays with Sum

Count the number of subarrays whose sum is equal to a given value using sliding window logic.

Real-World Use Cases

- Network traffic analysis – processing data over time intervals.
- Sensor data monitoring – analyzing trends in real-time.
- Streaming platforms – calculating watch time or behavior over recent intervals.
- Stock price analysis – moving average over recent days.

Q34. Subarray sum equal to K using hashing

```
class SubarraySumEqualsK {
   static int subarraySum(int[] nums, int k) {
      Map<Integer, Integer> map = new HashMap<>();
```

```
        map.put(0, 1);
        int sum = 0, count = 0;
        for (int num : nums) {
            sum += num;
            count += map.getOrDefault(sum - k, 0);
            map.put(sum, map.getOrDefault(sum, 0) + 1);
        }
        return count;
    }
}
```

Time: O(n), Space: O(n)

Q35. K-most frequent elements using priority queue

```
class KMostFrequent {
    static int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int num : nums) map.put(num, map.getOrDefault(num, 0) + 1);
        PriorityQueue<Map.Entry<Integer, Integer>> pq = new PriorityQueue<>((a, b) ->
a.getValue() - b.getValue());
        for (Map.Entry<Integer, Integer> entry : map.entrySet()) {
            pq.offer(entry);
            if (pq.size() > k) pq.poll();
        }
        int[] result = new int[k];
```

```
        for (int i = k - 1; i >= 0; i--) result[i] = pq.poll().getKey();
        return result;
    }
}
```

Time: O(n log k), Space: O(n)

Q36. Generate all subsets of a given array

```
class Subsets {
    static List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(result, new ArrayList<>(), nums, 0);
        return result;
    }

    static void backtrack(List<List<Integer>> result, List<Integer> temp, int[] nums, int start) {
        result.add(new ArrayList<>(temp));
        for (int i = start; i < nums.length; i++) {
            temp.add(nums[i]);
            backtrack(result, temp, nums, i + 1);
            temp.remove(temp.size() - 1);
        }
    }
}
```

Time: O(2^n), Space: O(n)

Q37. All unique combinations that sum to a target

```
class CombinationSum {
   static List<List<Integer>> combinationSum(int[] candidates, int target) {
      List<List<Integer>> result = new ArrayList<>();
      backtrack(result, new ArrayList<>(), candidates, target, 0);
      return result;
   }

   static void backtrack(List<List<Integer>> result, List<Integer> temp, int[] candidates, int remain, int start) {
      if (remain == 0) result.add(new ArrayList<>(temp));
      else if (remain > 0) {
         for (int i = start; i < candidates.length; i++) {
            temp.add(candidates[i]);
            backtrack(result, temp, candidates, remain - candidates[i], i);
            temp.remove(temp.size() - 1);
         }
      }
   }
}
```

Time: O(2^n), Space: O(n)

Q38. Generate all permutations of a given array

```
class ArrayPermutations {
    static List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        backtrack(result, new ArrayList<>(), nums);
        return result;
    }

    static void backtrack(List<List<Integer>> result, List<Integer> temp, int[] nums) {
        if (temp.size() == nums.length) {
            result.add(new ArrayList<>(temp));
        } else {
            for (int i = 0; i < nums.length; i++) {
                if (temp.contains(nums[i])) continue;
                temp.add(nums[i]);
                backtrack(result, temp, nums);
                temp.remove(temp.size() - 1);
            }
        }
    }
```

Time: O(n!), Space: O(n)

Q39. What is the Difference Between Subsets and Permutations? Provide Examples.

Subsets and Permutations are two fundamental concepts in combinatorics, but they are used in different contexts and have different meanings:

🔹 Subsets

A subset is any combination of elements from a set, regardless of their order.
The key idea is: order doesn't matter.

✅ Characteristics:

- Every element can either be included or excluded.
- A set with n elements has 2^n subsets (including the empty set).
- Example: For the array [1, 2], the subsets are:
  [], [1], [2], [1, 2]

📌 Used In:

- Power set generation
- Solving knapsack problems
- Finding combinations of items
- Backtracking-based problems

🔹 Permutations

A permutation is an ordered arrangement of elements.
Here, order matters.

✅ Characteristics:

- The number of permutations of n unique elements is n! (factorial of n).
- If selecting r elements out of n, the number of permutations is nPr = n! / (n - r)!
- Example: For the array [1, 2], the permutations are:
  [1, 2], [2, 1]

📌 Used In:

- Problems involving arrangement/sequencing
- Scheduling, puzzles (like Sudoku), pathfinding
- Backtracking and recursive tree-based problems

✅ Key Differences

Feature	Subsets	Permutations
Order Matters	❌ No	✅ Yes
Duplicates	Not allowed (unless specified)	Not allowed unless specified
Total Count	2^n	n!
Example ([1,2])	[], [1], [2], [1,2]	[1,2], [2,1]

Q40. Element with maximum frequency in array

```
class MaxFrequencyElement {
   static int maxFrequency(int[] nums) {
      Map<Integer, Integer> map = new HashMap<>();
      int maxCount = 0, res = nums[0];
      for (int num : nums) {
         map.put(num, map.getOrDefault(num, 0) + 1);
         if (map.get(num) > maxCount) {
            maxCount = map.get(num);
            res = num;
         }
      }return res;
   }
}
```

Time: O(n), Space: O(n)

Q41. Program to Find the Maximum Subarray Sum using Kadane's Algorithm in Java

```
public class KadaneAlgorithm {

    public static int kadane(int[] arr) {
        int maxSoFar = Integer.MIN_VALUE;
        int maxEndingHere = 0;

        for (int num : arr) {
            maxEndingHere = Math.max(num, maxEndingHere + num);
            maxSoFar = Math.max(maxSoFar, maxEndingHere);
        }

        return maxSoFar;
    }

    public static void main(String[] args) {
        int[] arr = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        System.out.println("Maximum subarray sum is: " + kadane(arr));
    }
}
```

Time Complexity: O(n), where n is the number of elements in the array.

Space Complexity: O(1), as we only use a constant amount of extra space.

Q42. Explain the Concept of Dynamic Programming (DP) and Its Use in Solving the Maximum Subarray Problem. Provide Examples.

🔷 What is Dynamic Programming (DP)?

Dynamic Programming (DP) is an algorithmic technique used to solve problems by breaking them into smaller overlapping subproblems, solving each subproblem only once, and storing the results (memoization) to avoid redundant calculations.

✅ Key Properties of DP Problems:

1. Overlapping Subproblems: The problem can be divided into subproblems that are reused multiple times.
2. Optimal Substructure: The optimal solution to the problem can be constructed from optimal solutions of its subproblems.

🔷 Application of DP: The Maximum Subarray Problem

The Maximum Subarray Problem involves finding the contiguous subarray (containing at least one element) that has the largest sum in a given array of integers.

✅ Kadane's Algorithm (A Classic DP Approach)

Idea:

- For each element at index i, we decide:
    - Either to include it in the existing subarray
    - Or start a new subarray from this element
- We keep two variables:
    - maxEndingHere → max subarray sum ending at current index
    - maxSoFar → overall max sum found so far

🔷 Kadane's Algorithm – Java Code Example

```
public int maxSubArray(int[] nums) {
```

```
    int maxEndingHere = nums[0];
    int maxSoFar = nums[0];

    for (int i = 1; i < nums.length; i++) {
        // Step 1: Either add to previous subarray or start new
        maxEndingHere = Math.max(nums[i], maxEndingHere + nums[i]);

        // Step 2: Update global max
        maxSoFar = Math.max(maxSoFar, maxEndingHere);
    }

    return maxSoFar;
}
```

◈ Example Input:

text

CopyEdit

Input: [-2, 1, -3, 4, -1, 2, 1, -5, 4]

Output: 6

Explanation: The maximum subarray is [4, -1, 2, 1], which sums to 6.

✅ Why Kadane's is Dynamic Programming?

- The solution to each subproblem (maxEndingHere[i]) depends on the solution to the previous subproblem (maxEndingHere[i-1]).
- We store intermediate results and use them to build the final answer.
- Avoids recalculating subarray sums repeatedly.

 Applications of DP (beyond Kadane's):

- Fibonacci numbers
- Longest Common Subsequence (LCS)
- Knapsack problems
- Matrix Chain Multiplication
- Edit distance, Coin change

Q43. Solve the Problem of Finding the Top K Frequent Elements in an Array

Algorithm:

1. **Count the frequency** of each element in the array.
2. Use a **min-heap** (or priority queue) to store the k most frequent elements.
3. Pop elements from the heap to maintain only the k most frequent elements.
4. Return the k elements from the heap.

Program:

```
import java.util.*;
public class TopKFrequentElements {
   public static List<Integer> topKFrequent(int[] nums, int k) {
      Map<Integer, Integer> countMap = new HashMap<>();
      for (int num : nums) {
         countMap.put(num, countMap.getOrDefault(num, 0) + 1);
      }

      PriorityQueue<Map.Entry<Integer, Integer>> minHeap =
         new PriorityQueue<>((a, b) -> a.getValue() - b.getValue());

      for (Map.Entry<Integer, Integer> entry : countMap.entrySet()) {
```

```
            minHeap.offer(entry);
            if (minHeap.size() > k) {
                minHeap.poll();
            }
        }

        List<Integer> result = new ArrayList<>();
        while (!minHeap.isEmpty()) {
            result.add(minHeap.poll().getKey());
        }

        Collections.reverse(result); // Optional to get descending order
        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1,1,1,2,2,3};
        int k = 2;
        List<Integer> result = topKFrequent(nums, k);
        System.out.println("Top " + k + " frequent elements: " + result);
    }
}
```

Time Complexity: O(n log k), where n is the number of elements in the array and k is the number of top elements we need to find.
Space Complexity: O(n), where n is the number of unique elements in the array.

Q44. Find Two Numbers in an Array that Add Up to a Target Using Hashing

Algorithm:

1. Iterate through the array.
2. For each element, calculate its complement (target - current element).
3. If the complement exists in the hash set, return the pair.
4. Otherwise, add the current element to the hash set.

Program:

```
import java.util.*;
public class TwoSum {
   public static int[] twoSum(int[] nums, int target) {
      Map<Integer, Integer> map = new HashMap<>();
      for (int i = 0; i < nums.length; i++) {
         int complement = target - nums[i];
         if (map.containsKey(complement)) {
            return new int[] { map.get(complement), i };
         }
         map.put(nums[i], i);
      }
      return new int[] {};
   }

   public static void main(String[] args) {
      int[] nums = {2, 7, 11, 15};
      int target = 9;
      int[] result = twoSum(nums, target);
      System.out.println("Indices of elements adding up to target: " + Arrays.toString(result));
```

```
    }
}
```

Time Complexity: O(n), where n is the number of elements in the array.
Space Complexity: O(n), as we are storing elements in a hash map.

Q45. Priority Queues and Their Applications in Algorithm Design

Priority Queue is a data structure where each element is associated with a priority, and the element with the highest (or lowest) priority is dequeued first. It is often implemented using heaps.

Applications in algorithm design:

1. **Dijkstra's shortest path algorithm** uses a priority queue to efficiently extract the minimum distance node.
2. **Huffman coding** for data compression uses a priority queue to build the optimal coding tree.
3. **Merge k sorted lists**: A priority queue can help merge multiple sorted lists in linear time.

Q46. Program to Find the Longest Palindromic Substring

Algorithm:

1. Use **expand around center** technique, where you check each character and expand outward to check for both odd and even length palindromes.
2. For each character, expand around it as the center and track the longest palindrome.

Program:

```
public class LongestPalindromicSubstring {


    public static String longestPalindrome(String s) {
        if (s == null || s.length() == 0) return "";


        int start = 0, end = 0;
```

```
        for (int i = 0; i < s.length(); i++) {
            int len1 = expandAroundCenter(s, i, i);  // Odd length palindrome
            int len2 = expandAroundCenter(s, i, i + 1);  // Even length palindrome
            int len = Math.max(len1, len2);

            if (len > (end - start)) {
                start = i - (len - 1) / 2;
                end = i + len / 2;
            }
        }
        return s.substring(start, end + 1);
    }

    private static int expandAroundCenter(String s, int left, int right) {
        while (left >= 0 && right < s.length() && s.charAt(left) == s.charAt(right)) {
            left--;
            right++;
        }
        return right - left - 1;
    }

    public static void main(String[] args) {
        String s = "babad";
        System.out.println("Longest palindromic substring: " + longestPalindrome(s));
    }
}
```

Time Complexity: O(n^2), where n is the length of the string.
Space Complexity: O(1), as we are using only a constant amount of extra space.

Q47. Explain the Concept of Histogram Problems and Their Applications. Provide Examples.

🔷 What Are Histogram Problems?

Histogram problems are a class of computational problems that deal with bar heights (usually represented as an array of integers), where each bar corresponds to a rectangle with a fixed width (usually 1 unit) and a variable height.

These problems often involve finding optimal areas, analyzing shapes, or determining capacity between the bars, and are commonly encountered in stack-based algorithm design.

🔷 Common Histogram Problems & Applications

1. Largest Rectangle in a Histogram

Problem:
Given an array of integers representing the heights of bars in a histogram, find the area of the largest rectangle that can be formed from contiguous bars.

Approach:

- Use a stack to keep track of indices of increasing bar heights.
- Time Complexity: O(n)

Example Input: [2, 1, 5, 6, 2, 3]
Output: 10
Explanation: Largest rectangle is between heights 5 and 6 → area = 5 × 2 = 10

---

2. Trapping Rain Water Problem

Problem:
Given an array of non-negative integers where each element represents the height of a bar, calculate how much rainwater can be trapped after raining.

Approach:

- Use two pointers or precomputed leftMax and rightMax arrays.
- Time Complexity: O(n)
- Space Complexity: O(1) (in two-pointer approach)

Example Input: [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: 6 units of water can be trapped between the bars.

Q48. Solve the Problem of Finding the Next Permutation of a Given Array

Algorithm:

1. Find the largest index i such that arr[i] < arr[i + 1]. If no such index exists, the array is the last permutation.
2. Find the largest index j such that arr[j] > arr[i].
3. Swap arr[i] and arr[j].
4. Reverse the subarray starting at arr[i + 1].

Program:

```
import java.util.Arrays;


public class NextPermutation {


   public static void nextPermutation(int[] nums) {
      int i = nums.length - 2;
      while (i >= 0 && nums[i] >= nums[i + 1]) i--;


      if (i >= 0) {
         int j = nums.length - 1;
         while (nums[j] <= nums[i]) j--;
         swap(nums, i, j);
      }
```

```
        reverse(nums, i + 1);
    }

    private static void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }

    private static void reverse(int[] nums, int start) {
        int end = nums.length - 1;
        while (start < end) {
            swap(nums, start, end);
            start++;
            end--;
        }
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        nextPermutation(nums);
        System.out.println("Next permutation: " + Arrays.toString(nums));
    }
```

Time Complexity: O(n), where n is the length of the array.
Space Complexity: O(1), since we modify the array in-place.

Q49. Find the Intersection of Two Linked Lists

Algorithm:

1. Find the lengths of both linked lists.
2. Align the two lists by skipping the extra nodes in the longer list.
3. Traverse both lists in parallel and return the intersection node when found.

Program:

```
class ListNode {
   int val;
   ListNode next;
   ListNode(int x) { val = x; }
}


public class IntersectionOfLinkedLists {


   public static ListNode getIntersectionNode(ListNode headA, ListNode headB) {
      if (headA == null || headB == null) return null;


      ListNode a = headA;
      ListNode b = headB;


      while (a != b) {
         a = (a == null) ? headB : a.next;
         b = (b == null) ? headA : b.next;
      }
```

```
        return a;
    }

    public static void main(String[] args) {
        ListNode headA = new ListNode(4);
        headA.next = new ListNode(1);
        ListNode intersection = new ListNode(8);
        headA.next.next = intersection;
        headB = new ListNode(5);
        headB.next = intersection;

        ListNode intersectionNode = getIntersectionNode(headA, headB);
        System.out.println("Intersection at node: " + intersectionNode.val);
    }
}
```

Time Complexity: O(n + m), where n and m are the lengths of the two linked lists.
Space Complexity: O(1), as we are using constant space.

Q50. Explain the Concept of Equilibrium Index and Its Applications in Array Problems. Provide Examples.

🔷 What is an Equilibrium Index?

An Equilibrium Index in an array is an index i such that the sum of elements to the left of i is equal to the sum of elements to the right of i.

Formally:

CopyEdit

A[0] + A[1] + ... + A[i-1] == A[i+1] + A[i+2] + ... + A[n-1]

🔷 Note: The element at index i itself is not included in either sum.

🔷 Example

text

CopyEdit

Input: A = [-7, 1, 5, 2, -4, 3, 0]

Equilibrium Index = 3

Explanation:

Sum of elements to the left of A[3] = -7 + 1 + 5 = -1

Sum of elements to the right of A[3] = -4 + 3 + 0 = -1

✅ How to Find Equilibrium Index (Efficient Approach)

1. Calculate the total sum of the array.
2. Initialize leftSum = 0.
3. Iterate through the array:
    - At index i, calculate rightSum = totalSum - leftSum - A[i]
    - If leftSum == rightSum, index i is an equilibrium index.
    - Update leftSum += A[i]

🔷 Time Complexity: O(n)

🔷 Space Complexity: O(1)

📌 Applications of Equilibrium Index

1. Partitioning Arrays:
    - Useful in problems where the array needs to be split into two parts with equal sums.
    - Can be used in game theory or balance problems.
2. Load Balancing:
    - In distributed systems, used to determine a balanced division of workload/data.
3. Algorithm Design:
    - Helps in solving problems related to prefix and suffix sums.
    - Basis for solving equilibrium points in circular arrays or linked lists.
4. Interview Questions:
    - Often asked in coding interviews as a test of understanding prefix sum, optimization, and linear scanning.